DEBRECENI EGYETEM
INFORMATIKA KAR

Csoportmunka- és feladatszervezési methodológiák

Témavezető:
Dr. Juhász István

Készítette:
Szacsúri László (MCAD)

Debrecen, 2007

1. Bevezetés

A Microsoft – bár a programozó társadalom tekintélyes része erről a tényről hajlamos megfeledkezni – igen régóta készít kifinomult és egyszersmind bonyolult alkalmazásokat a szoftverpiac számára.

Megannyi; akár néha ezer tagot is számláló – fejlesztői csapat hoz ki időről időre újabb és újabb verziót egy adott termékből, temérdek forráskódot tartva karban, nem tévesztve szem elől az egységes arculatot és a homogén *API*[1] struktúrát, mint amit például a *WCF, WF és a WPF*[2] követ.

A laikus is – az igazi szakmabeli pedig méginkább – elámul a számok hallatán; hogyan lehet 10 ezer programozó munkáját összehangolni?

Mégis, a közélet ezeket a teljesítményeket szóra sem méltatja; inkább azon csámcsog, hogy például a Vista hány éve is késik pontosan, és hogy miért kell nekünk egyátalán DirectX 10 egy ablakkezelőhöz...

Az ilyen szkeptikus kérdésekre a választ soha nem úgy kell keresnünk, hogy egy adott személyt teszünk egy funkció felelősévé; tehát mondjuk egy Apple termék esetén állandó bűnbaknak kikiáltjuk Steve Jobs-ot...

A szoftvernek azon felül, hogy egyesek szerint lelke is van, egy nagyon fontos meghatározója az *életciklus*a. Ha a ciklus adott periódusában nem állnak rendelkezésre a részfolyamat számára meghatározó és egyszersmind elengedhetetlen követelmények; a teljes szoftver életképtelen lehet, vagy csak igen hosszú vajúdás árán születik meg, s korántsem bizonyos, hogy azt nyújtja, amit vártak tőle.

Legyen szó akár egy mammutcégről, mint az MS vagy az Oracle, akár egyről a számtalan ügyeskedő magyar kényszervállalkozás közül, egy gyártó termékét mindig érdemes az alapján megítélni, hogy *kiknek*-, *kik*-, és hogy *hogyan* készítették.

[1] Application programming interface
[2] Windows Communication Foundation, Windows Workflow Foundation, Windows Presentation Foundation - Microsoft .Net Framework 3.0 - http://www.netfx3.com

A *„kinek?”* kérdésre a válasz bár egyértelműnek tűnhet, valójában nem csak a mindenkori fogyasztót jelenti. Nem mindegy, hogy a felhasználó valóban azt várja-e a terméktől, mint amit a fejlesztők gondoltak. Egy ilyen differencia persze adódhat akár kultúrális különbségekből is – a célközönség felmérése tehát alapvető feladat, mely kétféleképpen történhet:

- a kliens cég – melynek a termék készül – már rendelkezik elképzeléssel dolgozóinak-, fogyasztóinak szokásairól, képzettségéről, és ezt közvetíti a fejlesztők felé.
- a gyártó saját maga végez(tet) piackutatást a felhasználók igényeinek feltérképezése érdekében.

„Kik a fejlesztők?”

Temérdek szoftver bukik el azon, hogy a programozó csapat összetétele nem megfelelő, a vezetőség vagy túl szorosan, vagy túl lazán tartja a gyeplőt. Folyamatos felügyeletre van-e szükség, vagy csak időnként tereljük az embereket vissza a kijelölt útra? Lehet-e beleszólása egy egyénnek a teljes alkalmazástervbe, vagy az architect(ek)[3] vési(k) kőbe azokat? Sok minden határozhatja meg a légkört a fejlesztés során; és ez rányomja a bélyegét a végső produktumra is.

Azt pedig, hogy *„hogyan fejlesztünk?”* még a rengeteg jó és rossz példa ellenére is szinte lehetetlen általánosan megválaszolni.

Szakdolgozatommal – melyben ezekre a kérdésekre keresek lehetséges válaszokat – igyekszem mégis egy vázlatát adni a napjainkban elterjedt-, vagy terjedőben lévő alkalmazás-fejlesztési, illetve alkalmazás-menedzsmentet segítő methodológiáknak, különös tekintettel az iteratív módszertanokra.

Egy komplett *megoldás*[*(4. fejezet)*] részleteivel, személyes tapasztalataim hozzáfűzésével próbáltam tovább konkretizálni – illetve itt-ott árnyalni is a képet – hogy több-e, vagy kevesebb sikerrel, döntse el a Tisztelt Olvasó...

[3] tapasztalt fejlesztő, szoftver-mérnök

2. Módszertanok

Egy szoftvercég életében ideális esetben elég hamar eljön az a pillanat, amikor a vezetés saját maga eszmél rá, hogy az *ad-hoc* feladatszervezés ugyan gyors projekt-indítást tesz lehetővé, viszont a termék életciklusa során több problémát vet fel, mint amennyit megold.

Gyakran előforduló menedzsment hiba például, hogy egy éppen futó projektről nem tudják megmondani, hogy mikor ér majd pontosan véget, mivel a kliens új ötletei állandóan kitolják a határidőket. Ez abban az esetben nem is jelent túl nagy problémát, ha a cég képes egyetlen adott termékre fókuszálni; viszont egy nagyobb szoftvercsomag, vagy több kliens esetén, a párhuzamos fejlesztés megkövetelné az egymásra épülő komponensek közötti szinkronizációs pontokat, s konkrét *timeline*[4] híján szinte bizonyos a teljes csomag zárásának késése.

Természetesen ez a timeline is csak akkor ér valamit, ha a kliens és a fejlesztést végző csapat elkötelezett a betartását illetően, s egyik oldal sem tér el a benne szereplő határidőktől.

Mivel egy elég komplex alkalmazás esetén a komponensek sorrendi- illetve párhuzamosított szervezési igen nehéz feladat is lehet; tervezést és komoly odafigyelést igényel a fejlesztési vezetők részéről.

Gyors – gyakran nem is tudatos – és egyszerű kezelési megoldás a feladatok egy adott fejlesztői csapathoz rendelése, és annak mikro-menedzsmentje. Azt, hogy ezen rész-projekteket hogyan érdemes menedzselni, a közelítési módok határozzák meg.

[4] idő-terv

2.1. Közelítési módok

Az általánosan használt – nem feltétlenül csak szoftverfejlesztéshez alkalmazható, a projekt menedzsmentben bevált – közelítési módok a következők lehetnek:

2.1.1. Tradicionális közelítési mód

Egy tradicionális, fázisos közelítés a befejezéshez vezető lépések sorrendjét határozza meg.
Ebben a közelítésben a projektet 5 különálló szakaszra bontják (4 projekt szakasz valamint az ellenőrzés) a tervezés folyamán:

- Projekt indítási szakasz

Begyűjtik az igényeket

- Projekt tervezési szakasz

Az igények alapján felépítik az ütemtervet

- Projekt végrehajtási szakasz

A fejlesztők implementálnak

- Projekt ellenőrzési és követési rendszer

A tesztelés és a UAT[5] folyik

- Projekt befejezési, zárási szakasz

A terméket véglegesítik

2.1.2. Kritikus lánc

A kritikus lánc a tradicionális kritikus út módszer bővítése. A projekt menedzsmentet kritizáló tanulmányok gyakran felvetik, hogy az alapvető PERT[6]-alapú modellek a mai nagyobb fejlesző cégeken belüli több-projektes környezetre nem jól alkalmazhatók. Ezek közül a legtöbb nagyon széles skálájú, egyidejű, nem-rutinszerű projekt. Komplex modelleket használva ezekre a projektekre (vagy részfeladatokra) kiderül, hogy sok esetben néhány hét alatt jelentős felesleges költség keletkezhet, ugyanakkor nagyon kicsi a mozgástér ennek elkerülésére. A projekt menedzsment szakértők inkább megpróbálnak különféle "könnyű-súlyú" modelleket megalkotni, mint például az Extreme Programming[7] és a Scrum[8] - technika. Az Extreme programming

[5] User Acceptance Test – a végfelhasználói elégedettség mérésére szolgáló reprezentatív teszt
[6] Program Evaluation and Review Technique - http://www.netmba.com/operations/project/pert
[7] XP - http://www.extremeprogramming.org

technika általánosításával más projektekre kialakult az extrém projekt menedzsment, amely kombinálható a folyamat modellezéssel és a emberi kölcsönhatás menedzsment elveivel.

2.1.3. Folyamat-alapú (agile)

Szintén új koncepció a projekt ellenőrzésben az együttműködés a folyamat-alapú menedzsmenttel. Ez a terület az érettségi modell használatán alapul, mint például a CMMI[9] és a SPICE[10] (ISO/IEC15504), melyek egyre sikeresebbek a nagyobb lélegzetű projektek terén is.
A folyamat alapú (agile) menedzsment megközelítés az emberi kölcsönhatás menedzsment elvein alapul, és a folyamatokat az emberi együttműködések szemszögéből vizsgálja. Ez a közelítési mód alapvetően eltér a tradícionális közelítési módtól; az *agile szoftver fejlesztés*i közelítés szerint egy projekt inkább relatíve kis feladatok sorozata, a helyzettől függően elképzelve és végrehajtva, mint egy előre eltervezett, teljes folyamat.

[8] Scrum - http://www.scrumalliance.org
[9] Capability Maturity Model Integration - képességek érettségi modelljének integrálása
[10] Software Process Improvement and Capability Determination - szoftver folyamatok fejlesztése és képesség meghatározása

2.2. Tervezési methodológiák

Ha egy fejlesztői csoport végül egy adott közelítési mód mellett dönt, és belső igénye, hogy a számára legmegfelelőbbnek tűnő módszertant vezesse be mondjuk egy pilot-projekt beindításával; elengedhetetlen, hogy már a projekt tervezésénél is a methodológia által diktáltak szerint járjon el. Tehát a tervezésben és az irányításban a módszertannak azonosnak kell lennie, összhangban kell maradnia a projekt kezdetétől a végéig.

Következzen itt a tervezésnél használható, alkalmazható tipikus módszertanok történeti áttekintése, természetesen a teljesség igénye nélkül...

- **Folyamatábrák (Flowcharting – 196x)[I]**

  Egy sematikus reprezentációját adjuk a folyamatnak (algoritmusnak) melyet végig követünk a tervezés és a fejlesztés során. Számtalan szabvány létezik a leíró formátumra; a leggyakoribb mégis talán az UML[11].

- **Struktúrált tervezés (Structured programming – 1969)[II]**

  A procedurális absztrakció kiterjesztése tervezési szintre. Izolált, lehetőség szerint külön adminisztrálható egységek létrehozását és a rendszertelen vezérlés elkerülését szorgalmazza (lásd: „GOTO considered harmful”[12]). Alaptétele, hogy a következő három alaplépés használatával minden feladat leírható; s ez a tervezési lépésekre is igaz:

  1. szekvencia: egy részfeladat megoldása, majd azt követően egy másik részfeladat megoldása
  2. szelekció: egyik, vagy egy másik részfeladat kiválasztása, majd elvégzése egy adott feltétel teljesülése alapján
  3. iteráció: egy részfeladat többszöri egymás utáni megoldása, amíg egy feltétel teljesül

- **Fentről-lefelé tervezés (Top-down programming - 1970)[III]**

  A rendszer lépésenkénti teljes megértése alapján indítja a fejlesztési folyamatot. Ameddig a tervezésben nem jutottunk el a rendszer legalább egy részének a részletes ismeretéig, nem kezdődhet el a tényleges kódolás. Ez ugyan hátráltatja a fő funkcionális egységek

[11] Unified Modeling Language
[12] Edsger Dijsktra cikke 1968-ból

integrációs tesztelését amíg a tervezés nem terjedt ki szinte a végállapotig; viszont gyorsabb indítást tesz lehetővé - prototípus alkalmazásokat hatékonyan gyárthatunk.

- **Jackson féle struktúrált tervezés (Jackson Structured Programming – 1975)**[IV]
  Az adatfolyamok és egy program szerkezeti felépítése között próbál analógiát teremteni a struktúrált programozási módszereket véve alapul. Általánosított tervezési modellként is alkalmazható, ha a három alaplépést (lásd: *Struktúrált tervezés*) kiegészíjük az *alapvető műveletek definiálásával*.
- **SSADM (Structured Systems Analysis and Design Methodology – 1980)**[V]
  Olyan *waterfall* modellt megvalósító keretrendszer, mely elkülönült egységekre osztja fel az információs rendszer fejlesztésének munkáit és hajlékonyan idomul a különböző feladatokhoz. Segít a követelmények pontos elemzésében és a tervezett rendszer specifikálásában. Lépései:
  1. Analízis
  2. Vázlatos üzleti specifikáció
  3. Részletes üzleti specifikáció
  4. Logikai adat-tervezés
  5. Logikai folyamat-tervezés
  6. Valós rendszer-tervezés
- **IE/IEM (Information Engineering - 1981)**[VI]
  Elsősorban adatelemzési és adatbázistervezési módszerként született meg, melynek célja egy csapat erőforrásainak – legyen az eszköz-, vagy ember – maximális kihasználása az üzleti modell teljes lefedésének érdekében. Lépései:
  1. Stratégiai tervezés (ISP)
  2. Vázlatos üzleti analízis (OBAA)
  3. Részletes üzleti analízis (DBAA)
  4. Üzleti rendszer tervezés (BSD)
  5. Technikai tervezés (TD)
  6. Kivitelezés
  7. Végátmenet

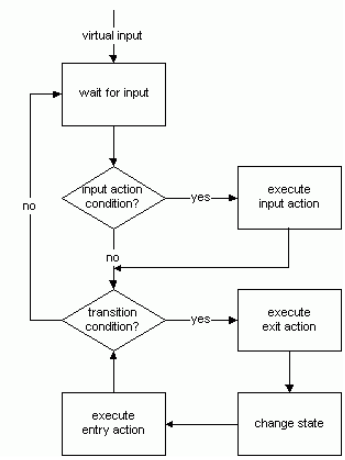
- **DSDM (Dynamic Systems Development Method – 199x)**

  Olyan keretrendszer, mely a RAD[13,VII] szemléletre épít, s azt iteratív-, valamint inkrementális megközelítéssel; állandó felhasználói visszacsatolással egészíti ki. Jobban képes reagálni az állandóan változó követelményekre, és jobb a kontrol a részfolyamatok felett. Fázisai:

  1. Elő-projekt: az induláshoz szükséges lépések vizsgálata
  2. A projekt életciklusa
     a. Megvalósíthatósági felmérés
     b. Üzleti felmérés
     c. Funkcionális modell iteráció
        i. Funkcionális prototípus azonosítása
        ii. Ütemterv megállapítása
        iii. Prototípus implementálása
        iv. Prototípus felülvizsgálata
     d. Valós iteráció tervezése és megépítése a funkcionális modell alapján
     e. Implementáció
  3. Utó-projekt: elkészült termék végellenőrzése, finomhangolása

- **VFSM (Virtual Finite State Machine – 1992)**

  Szoftver-specifikációs módszer, mellyel egy irányítási rendszer viselkedését írhatjuk le *bemeneti vezérlő szavak* illetve *kimeneti hatások* definiálásával. Az így előállított automatát használjuk a feladatok feldolgozására.



[13] Rapid Application Development

- **Scrum (1993)**[VIII]

  Egyszerű iteratív folyamatvezérlést és azonnali eredményességet nyújtó agile-módszertan. A neve a rugby-ben használt kifejezést tükrözi, mikoris a játékosok egymásba kapaszkodva viszik előre a labdát a cél felé. Előnye, hogy a csapat „saját magát szervezi", projekt-menedzsment szempontjából így tehát csak akadály-mentesítést kell végezni. A módszertan alapelve, hogy az igények és a rájuk adott megoldások analóg módon vannak definiálva, azonos tulajdonságokkal rendelkeznek. A főbb munkadarabok;

  - *Sprint*

    Az adott iterációt jelenti, tervezés során előre meghatározott kezdete és vége van.

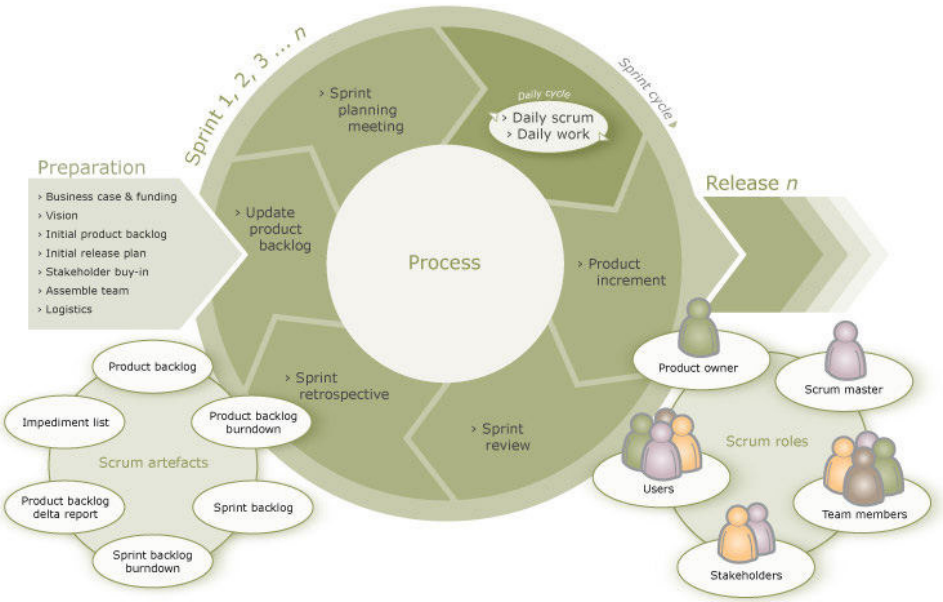
  - *Product backlog*

    Az igénylistát testesíti meg, ezt csak a Product owner módosíthatja.

  - *Sprint backlog*

    Az éppen aktuális sprinthez rendelt munkadarabokat tartalmazza. Elemei a következők lehetnek:

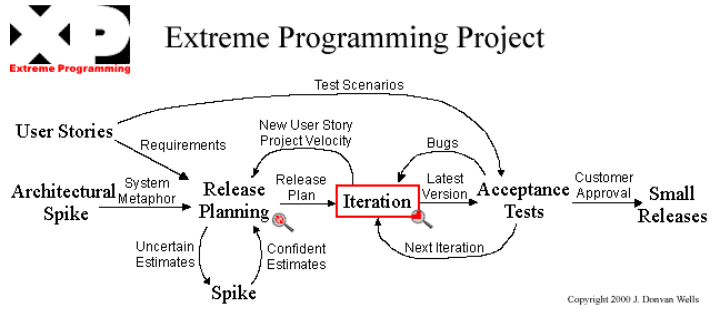
    - *Task* – a végzendő feladat
    - *Bug* – egy felismert hiba
    - *Impediment* – egy külső, a rendes folyamatmenetet akadályozó körülmény.



- **OODPM (Object Oriented Design using Prototype Methodology - 1994)**
  Olyan, az objektum orientált paradigma mentén felépített tervezési módszer ahol a prototípus-szemlélet is jelen van. Mind a folyamatok, mind az erőforrások, mind pedig a követelmények homogén rendszert alkotnak; az alapvető építőkő az objektum.
- **MSF (Microsoft Solution Framework - 1994)**
  Ügyfélközpontú, minőségorientált, sajátos csapatmodellel rendelkező módszertan, folyamatmodellje iteratív, adaptív és inkrementális spirálmodell. Alapvető célkitűzései:
  - Nyílt kommunikáció elősegítése
  - Csapatszellem fenntartása
  - Közös felelősségrevállalás és számonkérhetőség
  - Koncentráció az *üzleti érték* szállítására
  - Agilitásra való törekvés, változások folyamatos követése, adaptálódás
  - Nagy hangsúly fektetése a minőségre
  - Hibaelemzés, folyamatos tanulás

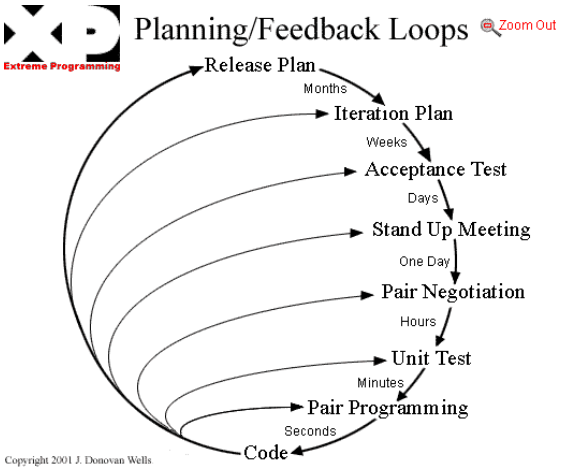
- **XP (Extreme Programming - 1996)**

  Az Extreme Programming folyamatmodelljét tekintve evolúciós jellegű feltáró fejlesztést végez. Rövid célbehatárolás és kezdeti felderítés után azonnal elkezdődnek az implementációt végző iterációk.



  Céljai a következők:

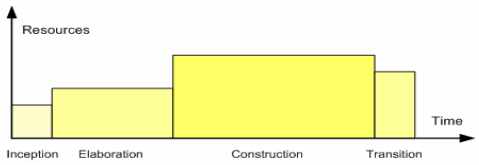
  - az emberközpontú- és a termelékenység-orientált vezérlés közötti határvonal elsimítása
  - a társas munkáról alkotott felfogás megváltoztatása (pár-munka)
  - a fejlődés elősegítése
  - egyedi fejlesztési stílus megteremtése
  - szoftver-fejlesztési szabályrendszer kialakítása



- **RUP (Rational Unified Process - 1998)**[IX]

  Iteratív szoftverfejlesztési methodológia, mely nem egy konkrét folyamatszervezési-előírást, hanem egy olyan keretrendszert specifikál, amelyből a fejlesztést végző csapatok a számukra legjobban alkalmazható elemeket válogathatják össze a teljes folyamat elvégzéséhez. Mint *business-driven development*[14], a RUP hat alapvetése:

  1. Alkalmazzuk rugalmasan a folyamatot
  2. Tartsuk egyensúlyban a prioritásokat
  3. Segítsük a csapatok közti együttműködést
  4. Demonstráljuk az eredményeket iteratívan
  5. Emeljük az absztrakciót a még kezelhető mértékig
  6. Fektessünk nagy hangsúlyt a minőségre



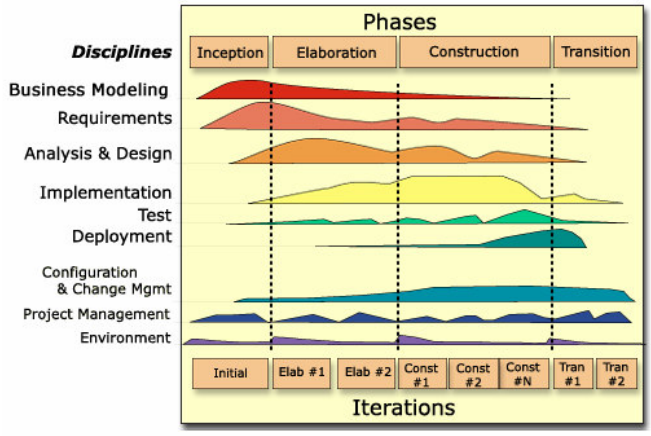
A projekt életciklusának állomásai itt a következők:

1. Kezdőállapot (Inception phase)

2. Kidolgozás (Elaboration phase)

3. Kivitelezés (Construction phase)

4. Átmenet (Transition phase)



[14] üzlet-vezérelt fejlesztési folyamat

3. A Team Foundation Server és a Visual Studio Team Suite

Mi is az a VSTS?

A Visual Studio Team Suite egy olyan termékcsomag, melynek részei együttesen lefedik a szoftver-termék fejlesztésének összes lépését; egységes felületet képezve a fejlesztői gárda minden szereplőjének számára.

Telepítéstől, licenszeléstől függően a csomag részei a következők lehetnek:

Microsoft Visual Studio Team Edition for Software Architects

Elosztott rendszerek tervezésére szolgáló (Distributed System Designer) felületet tartalmaz, amely beépített ellenőrző mechanizmusokkal rendelkezik, mint például a hálózati végpontok kompatibilitásának vizsgálata, vagy a szoftverkomponensek verziófüggőségeinek kényszerítése. A logikai adattárházak tervezésére szolgáló eszköz (Logical Datacenter Designer) további absztrakciót tesz lehetővé tervezési szinten, s segít a rendszer skálázhatóságának határait feltérképezni.

Microsoft Visual Studio Team Edition for Database Professionals

Egy olyan adatbázis projekt-típust kezel, melynek segítségével a szkriptjeink, lekérdezéseink is a forráskezelőben tárolhatók, a megszokott VS IDE-ben dolgozhatunk az adatbázisokon is. Egyszerű importálást biztosít már létező adatbázis sémából, valamint automatikusan generálhatóak segítségével az un. update-scriptek is. Támogatja továbbá a refaktoringot és referenciafüggőségek felderítését adatbázisokban is.

Microsoft Visual Studio Team Edition for Software Developers

Integrált statikus elemző eszköze segít a szoftver kódjában található esetleges biztonsági és teljesítményproblémák korai felismerésében. Automatikus *egység-tesztek* [15] generálását teszi lehetővé, melyeknek futtatása közben akár kódterület-lefedettséget is vizsgálhatunk.

Microsoft Visual Studio Team Edition for Software Testers

Automatikus-, illetve adat-vezérelt *egység-teszteke*t készíthetünk segítségével, valamint ezek szerkezetbe foglalásával komplett rendszer-terhelési tesztek írhatóak.

[15] unit test

Microsoft Team Test Load Agent

A Tester Edition által létrehozott rendszer-terhelési tesztek futtatására szolgál, melynek során akár 1000 aktív felhasználót képes szimulálni processzoronként. Webes alkalmazások stressz-tesztjéhez hasznos teljes virtuális böngésző-emulációt támogat.

Microsoft Team Foundation Server

A Team Suite alapjául szolgáló infrastruktúrális rendszert testesíti meg a következő szolgáltatásokat megvalósítva:

- Verziókezelő rendszer
- Munkadarab-követő rendszer
- Jelentésgenerátor
- Szerkesztő rendszer
- Kommunikációs platform

A TFS telepítéséhez szükséges kiegészítő rendszerek a következők:
Microsoft SQL Server 2005 Standard
Microsoft SQL Server 2005 Reporting Services
Microsoft SQL Server 2005 Analyzing Services
Microsoft Sharepoint Services 2.0
Microsoft Internet Information Server 6.0

Ezen fejezet további részeiben a Team Foundation Server szolgáltatásairól lesz szó.

3.1. Forráskezelés

Vitathatatlanul az egyik legfontosabb és szinte elkerülhetetlen feladat egy csoport hatékony munkájának biztosításakor, hogy a forrásállományokat, melyeken dolgoznak, egy mindenki számára elérhető, központi helyen is tároljuk el. Így az egyes fejlesztők lokálisan a forrásfa csak egy adott részével dolgoznak a saját gépükön, egymástól függetlenül javítva – vagy akár rontva el – a kódot. Ha erre a csapaton belül születik megállapodás; mindig csak a már rendesen működő, kitesztelt kód kerülhet be a forráskezelőbe, ez által biztosítva az átláthatóságot, a minőséget, és azt, hogy a termékből szinte bármikor készíthessünk egy „pillanatfelvételt", amit akár a megrendelőnek is megmutathatunk.

A Team Foudation Server egyik legnagyobb újdonsága az a szinte minden igényt kielégítő forrás- és verziókezelő rendszer, mely a Visual SourceSafe-t (VSS) hivatott végre leváltani a Microsoft által támogatott fejlesztőeszközökben.
Az új rendszer egy az alapoktól teljesen újraírt platformra épül, melynek funkcióit ugyanúgy *web service*[16]-k segítségével érhetjük el, mint a TFS többi alapszolgáltatását. Mivel a sztenderd HTTP kérések a legtöbb tűzfalon, proxyn akadálytalanul jutnak át, és lehetőségünk van SSL certifikátummal megtámogatni a kapcsolatot; biztonságosan érhetjük el a tárházat akár jelentős földrajzi távolságról is.
A szabványos MSSCCI[17]-t megvalósító illesztő – amelyhez SDK-t is adott ki a gyártó – lehetővé teszi, hogy bármilyen – akár open source – fejlesztői környezetben is használhassuk a TFS-t forráskezelőként. A TeamPlain Java Eclipse pluginja[18] az egyik legjobb példa erre.

Maguk a forrásállományok egy SQL Server 2005 adatbázisban tárolódnak, ezért azok migrálása, replikálása, mentése, és visszaállítása a teljesen szokványos adatbázis karbantartási feladatokkal megegyező módon történhet.

[16] Webes szolgáltatások – az MS értelmezése szerint majdnem mindig SOAP service-k értendők alatta
[17] Microsoft Source Code Control Interface
[18] http://www.devbiz.com/teamplain/eclipse/

3.1.1. Műveletek, művelet-halmazok

Érdekes koncepciót tükröző, és a TFS-en belül megvalósított tranzakciókezelés sajátosságait legjobban jellemző egység: a *changeset*[19].

Bármely a felhasználó által indítható művelet ugyanis csak changeset részeként létezhet, éppen ezért az olyan atomi műveletek sorozata mint a hozzáadás, módosítás, összefűzés - ideértve még a törlést is – egyszerre-, egy tranzakció részeként véglegesíthető, vonható vissza, avagy helyezhető a polcra az igények szerint.

Egy-egy ilyen változás-halmazhoz hozzárendelhetünk egy *munkadarab*ot is, miáltal később nyomon követhető, hogy az adott fejlesztő, aki a módosításokat eszközölte, milyen feladaton dolgozott éppen, lett légyen szó hibajavításról, vagy új funkció hozzáadásáról.

A changesetek listázhatóak dátum szerint, vagy visszakereshetők azonosító alapján, sőt; akár egy adott felhasználó egy bizonyos könyvtáron és egy adott időintervallumon belüli összes módosítását is lekérdezhetjük.

Ez megteremti a valós kódműveletek – az implementáció – és a folyamatirányítási rendszer közötti szoros kapcsolatot, melynek segítsével állandó, pontos képet kaphat a menedzsment, de a fejlesztők is, hogy hogyan halad az adott projekt, mennyi munka van még hátra, mennyire optimális a fejlesztői erőforrások kihasználtsága, stb...

Mivel a forrásfa elemeinek verzióit a changesetek határozzák meg, így a forráskezelőből történő adott verzió lekérése is történhet a fentebb felsorolt kritériumok alapján. Természetesen létező alapművelet a legfrissebb verzió lekérése is, melynek azonban van néhány sajátossága amiatt, hogy a forráskezelő megengedi a *nem exkluzív kijelölés*[20]-t is.

Amikor egy fejlesztő egy adott forrásfájlon megkezdi a munkát, *kijelöli*[21] *(„kicsekkeli")* az adott állományt – tudatja a forráskezelővel, és rajta keresztül a többi fejlesztővel is – hogy a fájlon dolgozni akar. Ezt a TFS-ben többféle módon teheti meg.

- ***Shared check-out*** – ez az alapértelmezett, és csak annyit jelez, hogy a fájlon dolgozik az illető
- ***Exclusive lock*** – ilyenkor a fájlt senki más nem csekkelheti se ki, se be.

[19] változás-halmaz
[20] none exclusive check-out
[21] check-out

- ***Exclusive check-in*** – megengedi, hogy a fájlt mások is kicsekkeljék, de be nem tudják csekkelni
- ***Exclusive check-out*** – megengedi, hogy a fájlt mások is becsekkeljék akiknél már kint van, de újabb fejlesztő már nem tudja kicsekkelni azt.

Furcsa – bár szintén logikus – sajátosság, hogy mivel a fájlok egyszerre több embernél is kint lehetnek módosítás alatt, ha valaki (egy harmadik személy) legfrisebb verziót kér a forrásból, nem a legutoljára becsekkelt – és esetleg konfliktuskezelésen is átesett – változatot fogja megkapni, hanem azt a verziót, ami azelőtt volt, hogy az első fejlesztő az adott fájlt kicsekkelte.
Másik érdekesség – szöges ellentétben például a VSS gyakorlatával – hogy egy állomány kicsekkelése nem vonja magával automatikusan a legfrissebb verzió lekérését is.
Amikor végül a programozók egyenként elkezdik becsekkelni a módosításaikat, a rendszer figyelmeztetést küld, majd elindítja a három utas konfliktus kezelőt[22]. Ennek segítségével egyszerre láthatjuk a saját verziónkat, a forráskezelő-szerveren lévő verziót, valamint a konfliktusok feloldása utáni lehetséges fájlverziót.
A változások és konfliktusok kezelése után azonban nem történik meg egyből a check-in, lehetőségünk van újrafordítani a kódot, vagy akár integrációs teszteket lefuttatni – így biztosítható a hibamentes *merge*[23]. Ez a folyamat viszont bármennyiszer önmagába fordulhat, hogyha a konfliktusok elhárítása közben valaki más „ránkcsekkel” – ilyenkor a rendszer ismét megkér minket a javításra. Ezt a szituációt szokták viccesen *merge*-lődésnek hívni...
A TFS forráskezelőjének felülete egy esemény-szolgáltatást is nyújt. Itt feliratkozhatunk például egy a forrásfa módosulásakor bekövetkező eseményre, egy adott interfészt megvalósító web-szolgáltatás metódusának beregisztrálásával.
Amikor egy fejlesztő becsekkel, bármilyen szükséges kiegészítő műveletet saját magunk implementálhatunk, akár egy teljesen más szerveren. Így összeköthető például a *build-szerver*(lásd: 3.2) is a forrásfával, tehát meg tudunk valósítani egyfajta folyamatos ellenőrzést. Ezt a mechanizmust szokás *constant integration*[24]-nek nevezni.

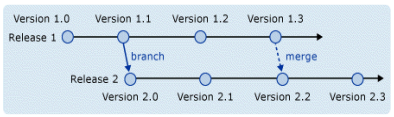
[22] bármilyen WinDiff-hez hasonlóan felparaméterezhető konfliktuskezelőre lecserélhető
[23] merge – összefésülés, konfliktuskezelés
[24] folyamatos integráció

3.1.2. Elágaztatás, visszavezetés

Amikor egy mérföldkőhöz ér a fejlesztés, vagy előre nem látható, hogy pontosan milyen irányba kell indulnuk egy funkció kifejlesztésének érdekében; szükséges lehet a forrásfa „*elágaztatása*”[25]. Az egyes komolyabb verziókezelők sokrétű keretrendszert nyújtanak a párhuzamosan dolgozó csoportok munkájának segítésére. Ezek fő funkciója, hogy biztosítsa egy-egy csoport számára a többiektől független kódterületet a forrásfában, ugyanakkor lehetővé tegye szinkronizációs pontok kezelését is. Bár forrásfáról beszélünk, legtöbbször a forrásfát a kód-elágazások miatt már sokkal inkább egy gráf jellemzi.

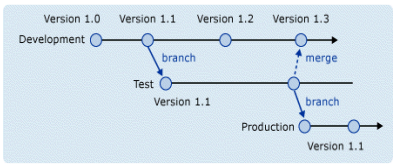
A következő stratégiák léteznek arra, hogy az elágazásokat mi alapján hozzuk létre:

- **Release alapon (Branch per Release)**



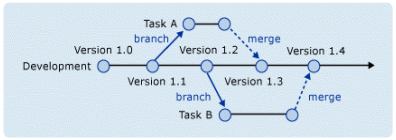
Ilyenkor az egyes verziók befejezésekor ágaztatunk el, az új verzió ága lesz a főág, és ha szükséges, a régebbi verziókat vezetjük vissza a főágba.

- **Kód-támogatás alapon (Code-Promotion Branches)**



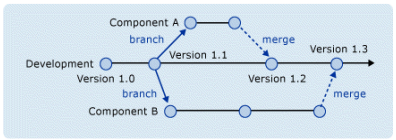
Célja, hogy elkülönítse a fejlesztési ágat, a tesztelési-kutatási ágat, és az élesben használt kód-ágat. A tesztágból nem csak visszavezetésre, de ágaztatásra is van lehetőségünk.

- **Feladat alapon (Branch per Task)**



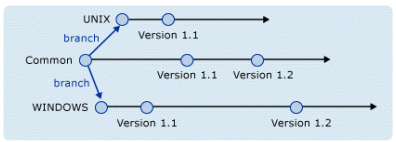
Minden egyes nagyobb feladatcsoport elvégzésére külön ágat hozunk létre, és csak sikeres kimenetel esetén vezetjük vissza a főágba a módosításokat.

- **Komponens alapon (Branch per Component)**



A komponens-alapú fejlesztést segíti, minden komponensnek saját ágat hozunk létre.

- **Platform alapon (Branch per Technology)**



Bár létezik közös metszete a különböző platformokat kiszolgáló kód-ágaknak, a főágba szinte sohasem vezetünk vissza.

[25] branching

3.2. Team build

Az integrációs tesztek bonyolítására és annak biztosítására, hogy a forráskezelőből bármikor szerkeszthető legyen egy futtatható, vagy telepíthető verziója az éppen fejlesztett alkalmazásnak; a legtöbb csapat saját build-szervert alkalmaz.

Olyan fejlesztési területeken, ahol egy specifikus hardverre írnak kódot, külön build-szervereket állítanak fel, így kihasználhatják az adott célhardver sajátosságait. A build-szerveren kialakítható ugyanakkor egy olyan teszt-környezetet is, melyben virtualizáció és/vagy emuláció használatával a különböző platformok modellezése is megoldható.

Egy-egy szerkesztés során azon felül, hogy a szerver a forráskódból tárgykódot generál, tranzakció-szerűen teszteket futtathatunk, telepítőcsomagot generálhatunk, vagy akár szalagra, CD-re archiváltathatjuk a *release*-ünket. A *build* csak akkor *sikeres*, ha az előírt összes lépése sikeresen futott le, minden egyéb esetben *hibás*nak minősítjük.

A buildet *constant integration* használata esetén a fejlesztők módosításai indukálják, viszont nagyobb lélegzetű projektek esetén ez túl sok erőforrást vihet el. Szokás ezért időszakosan; mondjuk napi, vagy heti buildet beidőzíteni. Természetesen az arra jogosult felhasználó bármikor manuálisan is indíthat buildet.

A minősítésnek leginkább itt jön elő a szerepe, mivel a TFS eszközrendszere lehetővé teszi a két build közötti kód-változások listázását is; így azonnal kideríthető, hogy melyik fejlesztő felelős a build „elhalásáért”.

Ez nem lebecsülendő kényszerítő erőt képez, és arra sarkallja a csapattagokat, hogy még körültekintőbben járjanak el a becsekkeléskor. Ha erre előzőleg megállapodás született, ám valaki mégis elnézett valamit, lehet a retorzió ilyenkor például a csoport közös kasszájába való befizetés is. Mivel ez a kassza többnyire a szórakozási célok financiális alapját képezi, így ez a módszer egyszerre fejleszti a csapatszellemet és tart fegyelmet a részlegen belül...

A buildek listázásakor kommunikációs szerephez is juthat, ha egy buildet speciális azonosítóval minősítünk. Mivel a TFS megengedi a minősítők testreszabását; a módszertanunknak, illetve a fejlesztői csapat felépítésének legmegfelelőbb azonosítókat használhatjuk.

Ha például napi rendszerességgel születnek a buildek, hasznos lehet egy adott mérföldkő elérésekor jeleznünk például a tesztelőknek, hogy az adott napi release már készen áll a tesztelési folyamatok megkezdésére.

Ugyanígy, ha a tesztelést végző csapat kritikus hibát talált, jelezheti a build megcimkézésvel, hogy van még mit csiszolni a kódon. Avagy, ha a tesztelés sikeresen véget ért, a release minősíthető például „*Signed Off*" jelzővel, így továbbkerülhet a disztibúcióért felelős csapathoz. Amint ők pedig értesültek arról, hogy az adott szoftver már bevezetésre vár, vagy akár már használatban is van, megjelölhetik a release-t „*In Production*"-re, és így tovább...

Ez utóbbi minősítőnek akkor van nagy szerepe, ha egy már élesben is használt terméken találnak hibát, melyet azonnal javítani kell. A verzió és a dátum ismerete alapján a forrásfa visszaállítható arra az állapotra, amiből a build készült. Így egy *gyors-javítás* [26] kiadása jelentősebb kockázatvállalás nélkül történhet meg, hiszen a kódnak csak azt a bizonyos részét kell módosítanunk ahol a hiba várhatóan volt, a többi maradhat érintetlenül.

[26] hotfix

3.3. Folyamatleíró sémák

A rengeteg kiegészítő szolgáltatáson felül, amitől egyértelműen több a Team Suite egy átlagos RAD rendszernél, az egy olyan folyamatvezérlést segítő; testreszabható integrált platform, mellyel mind a munkafolyamatokat, mind pedig a folyamat szereplőit és azok kapcsolatait leírhatjuk, újraértelmezhetjük – akár projektenként különböző szabályrendszert alkotva.

Ez azt jelenti, hogy saját magunk implementálhatunk szabványos módszertanokat megvalósító sémákat, és egy-egy új projekt létrehozásakor a természetének legmegfelelőbb módszertant választhatjuk ki.

A szabályrendszert maga a TFS tárolja *csapat-projekt*[27]-enként, ezáltal az egyes fejlesztői eszközök módosítása nélkül bármely módszertan különféle lépései ugyanazon környezetben használhatók.

A Team Suite mindegyik komponensében megtalálható a csoportmunka biztosítására szolgáló *Team Explorer* (lásd Függelék 1.) nevű eszköz. Feladata a folyamatos kommunikáció fenntartása a csapattagok között, valamint hozzáférést biztosít a TFS minden alapszolgáltatásához. A munka-folyamatok kezelése, a portál-integráció, a jelentés-kezelő, és a team-buildek futtatása is innen történhet, valamint segítségével adminisztrálhatók a jogosultsági körök-, illetve beállítások minden egyes csapat-projektre. (lásd Függelék 2.) A megfelelő jogkörrel rendelkező személy innen elérheti a folyamatleíró-sémák menedzselését megvalósító eszközt is.

A sémakezelő legfontosabb tulajdonsága, hogy minden adott azonosítóval ellátott folyamatleíró sémához egy éppen aktuális alap-verziót társít, s amikor új projektet hozunk létre, ennek egy klónját rendeli ahhoz. Így minden egyes csapat-projekthez képes tárolni az alap-verziónak egy csak az adott projektre jellemző társ-verzióját, amit a későbbiekben akár menet közben is változtatgathatunk.

Egy-egy sémát gyakorlatilag XML fájlok segítségével definiálunk, ami a jól szervezett hierarchia könnyű áttekintését és egyszerű szerkesztését segíti elő.

A séma alapkönyvtárában a következő tematika szerint kell kialakítanunk alkönyvtárakat a folyamatvezérlő komponenseknek megfelelően:

[27] Team project – a TFS terminológiájában az egy adott termékhez rendelt munkaterületet jelenti

Classification – Osztályozás

Az MS Project integráció biztosításához adhatunk meg leképezéseket a munkadarabok tényleges mezői és a csatolófelület között.

Global lists – Globális listák

Itt definiálhatóak azok a láthatóságukat tekintve globális – az összes csapat-projekt számára elérhető – kulcs-érték párok, melyeket a sémában hivatkozhatunk, felhasználhatunk például validációra, választómezők értéklistájának populálására.

Groups and permissions – Csoportok és jogosultságok

Az egymásba ágyazható jogosultsági körök definiálására szolgál, ahol mind a TFS szerveren már létező globális jogköröket, mind az adott csapat-projektben használható lokális csoportokat adhatjuk meg.

Reports – Jelentések

Ebben a szekcióban a Reporting Services számára adhatók meg előre definiált lekérdezések az RDL leíró nyelv segítségével. A csapat-portálon (lásd Függelék 4.) elérhetővé tehetők ezek a lekérdezések, s szerepük a projekt állásának valós idejű megjelenítésében, a lehetséges hibák transzparensé tételében van.

Version control – Verziókezelő

A forráskezelő alapállapotát szerkeszthetjük ebben az alkönyvtárban, itt rendelhetünk a jogosultsági csoportokhoz olvasási, írási, és egyéb jogokat a forrásfa eléréséhez.

Workitem tracking – Munkadarab követés

A legfontosabb alkönyvtár, melyben a munkadarabokat és azok kapcsolatait adhatjuk meg a típusdefiníciók leírásával. Ugyanitt nyílik lehetőség a munkadarabok szerkesztőnézeteinek importálására is.

Windows Sharepoint Services – WSS

A WSS (csapat-)portál alapállapota állítható itt be, megadhatók azok a dokumentumsablonok, melyek segítségével a munkadarabok majd MS Office termékek használatával nyomon követhetőek, adminisztrálhatóak.

4. A Scrum módszertan implementációja

Ebben a fejezetben a Scrum módszertannak egy a TFS rendszerre alkalmazott saját megvalósítását szeretném bemutatni, melynek alapjául a Conchango és a Scrum Alliance által összeállított *Scrum for Team System 1.0* című séma szolgált.

A lehetőségekhez mérten próbáltam megtartani a módszertan esszenciális egyszerűségét, az eszközrendszert felhasználva segíteni a mindennapi munkát, és nem állandó gátak közé szorítani a fejlesztőket, mint azt a legtöbb hasonló, a gyakorlatban is alkalmazott csomag teszi.

Vannak természetesen olyan részei, melyek további automatizálást igényelnének, viszont alapjában véve egy használható, kényelmes keretrendszert nyújt a programozók számára. Mivel az adminisztratív teendőket a rendszer részint a kód változásának implikációjaként végzi, részint pedig a Scrum master veszi át ezeket a terheket; a fejlesztő produktivitása tovább növelhető, így a legfontosabb feladatára koncentrálhat; magára a kód írására, tesztelésére.

A járulékos tevékenységek minimalizálásán túl, ennek a sémának az egyik erőssége, hogy biztosítja a nyugalmat a fejlesztőknek az iteráció közben is, megakadályozva, hogy a kitűzött cél csak délibábként lebegjen a csapat szeme előtt, mint ahogyan az gyakran előfordul, ha a kliens újabb és újabb igénnyel él a termék funkcionalitásai iránt.

Mivel a methodológia iteratív, magát a sémát is a módszer alkalmazásával alakíthattam ki; folyamatosan csiszolva a komponenseken, így a frissebb verziók mindig a leggyakrabban használt felületeken változtak a legtöbbet.

4.1. Szerepkörök

A Scrum módszertan - mivel a könnyű-súlyú közelítési módot segíti elő – csak viszonylag kevés szerepkört definiál, és azok tagjai is dinamikusan allokálhatók az éppen aktuálisan felmerülő igények szerint.
A számonkérhetőség, illetve a bizalom az egyes rétegek között egy *közös megállapodás*[28] alapján biztosítva van, így szinte nem létezik hierarchia a szereplők között, egyátalán nincsenek alárendelt szerepek.

4.1.1. Product owner

A valós fejlesztést végző csapathoz nem szorosan kötődő szerepkör, melyet rendszerint egyetlen személy tölt be a *Product owner*-é. Ő felelős a kliens által felvetett igények összegyűjtéséért, rendszerezéséért, priorizálásáért. Természetesen ebben a módszertanban ez a szerepkör egyfajta absztrahált klienst testesít meg, viszont sokkal nagyobb tapasztalattal rendelkezik a csapat tevékenységét illetően, mint a megrendelő. Hatékonyabbá teszi az oda-vissza kommunikációt, valamint elsődleges szakmai szűrőként is funkcionál, amennyiben egy *business analyst*[29] tölti be ezt a szerepkört.

4.1.2. Scrum master

A Scrum-team névleges felelős vezetője a *Scrum master*, akinek elsődleges célja a csapat tagjai előtt felmerülő akadályok megszüntetése, a tagok közötti esetleges nézeteltérések kezelése, és a belső emberi igények továbbítása a management felé. A napi rendszerességgel zajló stand-up meetingen elsősorban ő a moderátor, a csapattagok neki válaszolják meg a sztenderd három kérdést:

- Mit értem el a legutóbbi alkalom óta?
- Milyen akadályokba ütköztem?
- Mit csinálok a következőkben?

[28] Common agreement
[29] Üzleti elemző

4.1.3. Team members

A Scrum szerinti csapat egy önszerveződő egység, mely nem igényel külső behatást, sem irányítást feladatainak megoldására. Éppen ezért eredményesség szempontjából akkor a legideálisabb az összetétele, ha az adott problémáról a legkülönbözőbb elképzeléssel rendelkező fejlesztők alkotják. Ugyanakkor egy homogén, mondjuk csak senior fejlesztőkből álló csapat az állandó véleménykülönbségekből profitálhat is, ám valószínűbb az állandó „harc" miatti alacsonyabb produktivitás.

Álljon itt egy sarkított példa a csapat összetételének fontosságáról, melyet sok helyen hivatkoznak Scrumról szóló publikációk.

„A tyúk és a disznó elhatározzák, hogy gyors-büfét nyitnak, és olcsó reggelit fognak felszolgálni a pitvar mellett elhaladó munkába igyekvő állatoknak... A menü összeállításánál azonban egyből összetűzésbe kerülnek; a csirke a sonkás-tojás mellett érvel, de a malac valahogy nem eléggé motivált..."

Bár mindkét fél számára kellemetlen az étel előállítása, a felek nem egyenlő arányban osztoznak a kockázaton. A Scrum csapat csak akkor sikeres, ha a tagjai ugyanolyan erőbedobással képesek dolgozni az előbbrejutás érdekében.

4.1.4. Shared resources

Elég fontos lehet egyes esetekben külön szerepkört rendelni az időosztásban dolgozó személyekhez, akiken több Scrum-csapat egyszerre osztozik. Ebbe a rétegbe sorolható például egy a rendszer integritásáért felelős senior architect, vagy mondjuk a dokumentációt szolgáltató tech-writerek. Ezek az emberek akár saját Scrum csapatot alkothatnak, és saját projektekként kezelhetik az egyes termékekhez nyújtott szakértői tevékenységüket.

4.2. Munkadarabok

Egy adott folyamat-leíró séma meghatározó építőköve a *munkadarab*[30].

A Scrum séma talán legfontosabb egysége az iterációt is definiáló *Sprint*. Egy sprinthez rendelhető *Product backlog item*-jeinek felbontása *Sprint backlog item-ekké* az, ami előállítja az aktuális *Spring backlog*-ot. Mivel minden munkadarabhoz rendelhető egy- vagy több másik munkadarab, így definiálhatók a függőségek az egyes darabok között. Valójában a folyamat egyes lépései-, hozzárendelései az operátorok, az operandusok pedig maguk a munkadarabok. A viselkedést, tehát a szemantikai szabályokat mindig maga a séma definiálja.

Az adott sémában használható munkadarab definíciókat a következő módon adhatjuk meg a séma gyökér-leíró fájlában:

```
<?xml version="1.0" encoding="utf-8" ?>
<tasks>
   <task
      id="WITs"
      name="WorkItemType definitions"
      plugin="Microsoft.ProjectCreationWizard.WorkItemTracking"
      completionMessage="WorkItemTypes created"
      completionDescription="Work item types used by work item tracking">
      <taskXml>
         <workItemTypes>
            <workItemType fileName="WorkItems\Types\Sprint.xml"/>
            <workItemType fileName="WorkItems\Types\ProductBacklogItem.xml"/>
            <workItemType fileName="WorkItems\Types\SprintBacklogItem.xml"/>
            <workItemType fileName="WorkItems\Types\TestCase.xml"/>
         </workItemTypes>
      </taskXml>
   </task>
</tasks>
```

Ahol az egyes munkadarabok típus-definícióját külön XML-alapú, úgynevezett WITD formátumú dokumentumokban adjuk meg.

A WITD fájlok a munkadarabok adatbázisbeli reprezentációját adják meg, tehát a mezőket írják le, azoknak típusaival, lehetséges értékeivel is számolva.

[30] Work item

Mivel a sémába új munkadarab típust csak az ős-munkadarab rekord kiegészítéseként vihetünk fel, az összes munkadarab rendelkezik az egy adott munkadarab típus számára definiált mezőkkel. Ez az öröklődési anomália az adatbázisbeli reprezentációból fakad, és sajnos néha ugyancsak zavaró, amikor például egy taszk típusnak létezik *Remaining Work*-je a *Sprint backlog item*-től és van *Capacity*-je is amit a *Sprint* definíciójától örökölt.

A munkadarabok őse a következő általánosan alkalmazható *default* mezőkkel rendelkezik:

Audit: flag-jellegű mező a változáskövetés biztosítására
Changed By: a legutóbbi módosítást végző neve
Changed Date: a legutóbbi módosítás dátuma
Closed By: a munkadarab lezárását végző neve
Closed Date: a munkadarab lezárásának dátuma
Created By: a munkadarab létrehozójának neve
Created Date: a munkarab létrehozásának dátuma
History: a munkadarab életciklusának teljes története
Owner: egy az adott munkadarab számára kijelölt felelős neve
Resolved By: a munkadarab megoldójának neve
Resolved Date: a munkadarab megoldásának dátuma
State Changed Date: a munkadarab legutóbbi státuszváltásának dátuma
Team Project: a munkadarab honos projektje

4.2.1. Sprint

A Sprint az az alapegység, amely az elvégzendő, elvégezhető munka mennyiségét határozza meg. Amint a megrendelőnek, vagyis a Product ownernek annyi igénye merül fel, hogy abból már egy 3-, 4-, esetlegesen 5 hétig tartó fejlesztési iterációt el lehet kezdeni, létrehozhatunk egy új sprintet. A sprint a *Sprint Planning Meeting*-gel kezdődik, ahol meghatározzuk a hosszát, a csapat tagjait, a tőlük elvárható maximális munkaóra-kapacitást, számolva a szabadságokkal is. Azt is itt kell eldönteni, hogy az adott sprint végén lesz-e release, vagy a sprint célja csupán egy prototípus előállítása.

Ezeket az értékeket a munkadarab életciklusa során sokszor használhatjuk majd, ezért mindet felvesszük a munkadarab típus-definíciójába a következő módon:

```
<WITD application="Work item type editor" version="1.0"
xmlns:witd="http://schemas.microsoft.com/VisualStudio/2005/workitemtracking/ty
pedef">
   <WORKITEMTYPE name="Sprint">
      <DESCRIPTION>Includes information about a Sprint</DESCRIPTION>
      <FIELDS>
         <FIELD   name="Title"
                  refname="System.Title"
                  type="String"
                  reportable="dimension">
            <HELPTEXT>This is the name and/or number of the Sprint</HELPTEXT>
            <REQUIRED />
            <CANNOTLOSEVALUE />
         </FIELD>
         <FIELD   name="State"
                  refname="System.State"
                  type="String"
                  reportable="dimension">
            <HELPTEXT>This is the status of the Sprint</HELPTEXT>
         </FIELD>
         <FIELD   name="Work Item Type"
                  refname="System.WorkItemType"
                  type="String"
                  reportable="dimension" />
         <FIELD   name="IsReleaseSprint"
                  refname="VSTS.Scrum.IsReleaseSprint"
                  type="String"
                  reportable="dimension">
            <HELPTEXT>Defines whether the Sprint produces a release</HELPTEXT>
            <ALLOWEDVALUES expanditems="true">
                  <LISTITEM value="No" />
                  <LISTITEM value="Yes" />
            </ALLOWEDVALUES>
            <DEFAULT from="value" value="No" />
         </FIELD>
```

```
            <FIELD   name="Sprint Start"
                     refname="VSTS.Scrum.SprintStart"
                     type="DateTime"
                     reportable="detail">
               <HELPTEXT>The date and time the sprint is due to start</HELPTEXT>
               <REQUIRED />
            </FIELD>
            <FIELD   name="Sprint End"
                     refname="VSTS.Scrum.SprintEnd"
                     type="DateTime"
                     reportable="detail">
               <HELPTEXT>The date and time the sprint is due to end</HELPTEXT>
               <REQUIRED />
            </FIELD>
            <FIELD   name="Capacity"
                     refname="VSTS.Scrum.Capacity"
                     type="Integer"
                     reportable="detail">
               <HELPTEXT>Number of hours of development work you have</HELPTEXT>
               <REQUIRED />
            </FIELD>
            <FIELD   name="Description"
                     refname="System.Description"
                     type="PlainText">
               <HELPTEXT>This is the full description of the goal</HELPTEXT>
               <REQUIRED />
            </FIELD>
            <FIELD   name="SprintID"
                     refname="VSTS.Scrum.SprintID"
                     type="String"
                     reportable="detail">
               <HELPTEXT>The Sprint (iteration) number</HELPTEXT>
               <REQUIRED />
               <CANNOTLOSEVALUE />
            </FIELD>
            <FIELD   name="Team Project"
                     refname="System.TeamProject"
                     type="String"
                     reportable="dimension" />
         </FIELDS>
         .
         .
         .
      </WORKITEMTYPE>
      .
      .
      .
</WITD>
```

Fontosabb tag; a mező-specifikációknál például a ***REQUIRED*** alakú. Ha ez meg van adva, az adott munkadarab módosításakor a mező értéket kötelezően ki kell tölteni.

Egy másik hasonló, a ***CANNOTLOSEVALUE***, ami arra szolgál, hogy ha már az adott rekord egyszer tartalmazott értéket egy mezőhelyen, az többet ne állhasson határozatlan, vagy üres állapotba.
Az ***ALLOWEDVALUES*** arra szolgál, hogy felsorolással vagy globális listára való hivatkozással megadhassunk egy listát, melynek elemei közül vehet fel csak a mező egyet értékül.

A ***DEFAULT*** tag segítségével a határozatlan állapotú mezők értékét állíthatjuk be egy fix értékre.

Egy mező szükséges attribútumai a ***name***, a ***refname***, a ***type***, és a ***reportable***. Ezeknek rendre a következő a szerepük:

- Egyértelmű, egyedi azonosító a mező elérésére
- Minősített név, ami a származtatott típust azonosítja
- A primitív érték-típust adja meg
- Az Analyzing Services által választott OLAP feldolgozási mód megadására szolgál

Minden munkadarab leíró állományban adhatunk meg úgynevezett *folyamat-átmenet* definíciót. Ennek feladata, hogy az adott munkadarab egyes mezőinek értékváltozásaira megkötéseket specifikáljon. Segítségével a legkönnyebb megvalósítani a munkadarab állapotának változás-követését; egy jó szabályrendszer segítségével az életciklusának állomásait is maradéktalanul leírhatjuk.

```
<WORKFLOW>
   <STATES>
      <STATE value="Not Done" />
      <STATE value="In Progress" />
      <STATE value="Done" />
   </STATES>
   <TRANSITIONS>
      <TRANSITION from="" to="Not Done">
         <REASONS>
            <DEFAULTREASON value="New" />
         </REASONS>
         <FIELDS>
            <FIELD refname="VSTS.Scrum.Audit">
                  <DEFAULT from="value" value="0" />
            </FIELD>
         </FIELDS>
      </TRANSITION>
```

```
      <TRANSITION from="Not Done"
                  to="In Progress"
                  for="[Project]\Scrum masters">
         <REASONS>
            <DEFAULTREASON value="Work has commenced " />
         </REASONS>
         <FIELDS>
            <FIELD refname="VSTS.Scrum.Audit">
                  <DEFAULT from="value" value="0" />
            </FIELD>
         </FIELDS>
      </TRANSITION>
      .
      .
      .
   </TRANSITIONS>
   .
   .
   .
</WORKFLOW>
```

Ahogyan az a ***TRANSITION*** elemek szerkezetéből is látszik, bármely állapotból (***from***) definiálhatunk átmenetet bármely másik állapotba (***to***). A rendszer csak az itt felsorolt állapot-váltásokat engedélyezi, így képezhetjük le azt a valós folyamatvezérlési módszert, mint amit például a Scrum diktál.

Megkötést alkalmazhatunk a ***for*** attribútum kitöltésével arra nézve is, hogy kizárólag mely jogosultsági csoportnak tesszük lehetővé az állapot váltását.

Tehát a sprint lehet kezdő-állapotban, lehet éppen folyamatban, és lehet befejezett is.

Mivel a folyamat leírása akár aciklikus gráf is lehet, egy adott állapotba el tudunk jutni több másik állapotból is. Ilyenkor hasznos a ***REASONS*** tag, melyben egy rövid szövegben rögzíthető az adott átmenet oka és célja is.

A ***FIELDS*** elem segítségével a munkadarab státuszának változásakor kísérő eseményként egy vagy több másik mező értékének módosítását írhatjuk elő. A *VSTS.Scrum.Audit* nevű mező változtatása itt az adminisztrációs háttérprocesszeknek szól – egy külső eseménykezelőt implementálva; elindul például a jelentés-generátor szemétgyűjtő mechanizmusa.

Természetesen definiálnunk kell egy szerkesztési felületet (lásd Függelék 3.) is az újonnan létrehozott munkadarabunk számára. Ezt például a következő módon tehetjük meg:

```
<Form>
   <Layout>
      <Group>
         <Column PercentWidth="100">
            <Control    FieldName="System.Title"
                        Type="FieldControl"
                        Label="Title"
                        LabelPosition="Left" />
            .
            .
            .
         </Column>
         .
         .
         .
      </Group>
      .
      .
      .
   </Layout>
</Form>
```

A munkadarab mezőit egy egymásba skatulyázott *control*-elemeket tartalmazó elrendezésben helyezhetjük el néhány alapvető formázási módosítót követve.

Az értékek különféle formátumban való bekérésére használható a *Control Type* attribútumában szereplő érték.

A *FieldControl* egy egyszerű szövegmező, mely a mező típusától függően – amennyiben kényszerfeltételeket definiáltunk a lehetséges értékeire nézve – választómezőként is funckionálhat.

Létezik például *HtmlFieldControl* is, amivel általában a hosszabb szöveges leírásokra szolgáló mezőket szokás megjeleníteni.

A *DateTimeControl* nevének megfelelően jól használható például a sprint kezdetének, illetve végének a beállítására.

Egy speciális control-fajta a *WorkItemLogControl*, mely egy csak olvasható nézetét adja a munkadarabon történt változások részletes listájának.

Megint másik kiegészítő funkcionalitással is bíró control az *AttachmentsControl*, amivel a munkadarabhoz külső állományokat csatolhatunk, rögzíthetünk.

4.2.2. Product backlog

A Product backlog az a lista, amelynek minden körülmények között tartalmaznia kell a megrendelő- illetve a belső architectúrális igények által támasztott követelmények részletes leírását, prioritását, és egy sor olyan egyéb tulajdonságot, melyek segítségével az *feature*-elem egyértelműen beazonosítható.
Mivel a *Product backlog*-ba – akár az iteráció közben is – állandó rendszerességgel érkezhetnek újabb elemek, és karbantartását a *Product owner* végzi, a szigorú értelemben vett Scrum-team csak a sprint kezdetekor találkozik vele.
A szakmai tervezést segítendő, a *Product backlog item* a rekord azonosítókon és az állapotleírókon felül a következő primitív mezőkkel kell hogy rendelkezzen:

Description – Leírás : a követelmény részletes, ám lényegre törő megfogalmazása
Priority – Prioritás: az üzleti oldal által meghatározott fontossági mérőszám. Értéke 1-től 5-ig változik, mindig a legkisebb élvezi a legnagyobb prioritást. Néhányan nagyobb felbontású skálát ajánlanak, ám a gyakorlatban teljesen elegendő az 5 féle árnyalat.
Risk – Rizikó: architectúrális, illetve fejlesztési szemszögből elemezve az adott követelményt, a csapat megállapíthat a prioritás mellé egy bizonytalansági tényezőt ugyanúgy 1 és 5 között, ennél viszont fordított az arány, az 5-ös a legnagyobb.
Estimated effort – Becsült költség: ismerve a csapat képességeit, és a már esetleg létező komponensek komplexitását; ez a munkaórákban kifejezett szám adja a bruttó fejlesztési költséget. A rizikó általában szorzóként funkcionál a költség meghatározásakor is, minél több sprintet élt meg együtt a csapat, az esztimációk annál pontosabbak lesznek.

A csapat a követelményekkel érdemben a *Spring planning meeting*-en kezd el foglalkozi, itt töltik ki az utóbbi 2 mezőt, és a sprintben elérhető bruttó kapacitást-, illetve az igények prioritását figyelembe véve választják ki a sprinthez rendelendő Product backlog listaelemeket.

4.2.3. Sprint backlog

A Sprint backlog egy némiképp ellentmondásos gyakorlati megnevezése annak a listának, amely az éppen aktuális sprinthez rendelt feladatokat[31], hibákat[32], illetve külső körülményeket[33] tartalmazza. A furcsasága abban rejlik, hogy a szakirodalom általánosan az összes előbb felsorolt elemet *Sprint backlog item*-nek nevezi, függetlenül attól, hogy ezek az elemek hozzá lettek-e rendelve a futó sprinthez vagy sem. Mégis az egyszerűség kedvéért ezt a terminológiát fogom alkalmazni a továbbiakban is.

Mivel egy-egy a *Product backlog*-ban található követelmény például *Non-Functional Requirement* is lehet, valamint a kliens nem feltétlenül számolhat az igényei kielégítéséhez szükséges fejlesztési költségek valós súlyával, a *Product backlog item*-ekhez mindig hozzárendelünk egy- vagy több *Sprint backlog item*-et. A sprint kezdetekor a követelményeknek konkrét feladatokká történő ilyen módú tesszelációja segíti az átláthatóbb, könnyebb szervezést, és a csapat munkáját. Egy-egy feladat, taszk nem lehet rövidebb mint 4 munkaóra, de maximálisan 20 fejlesztői órát allokálhatunk rá.

A Scrum filozófiája alapján, amikor a csapat előállította a sprint tervezéséhez szükséges aktuális *Sprint backlog*-ot, megkezdődhet az egyes taszkokra való önkéntes jelentkezés. A Scrum master természetesen felülbírálhatja az egyének döntését, illetve javasolhat az egyes területeken nagyobb tapasztalattal rendelkező fejlesztők számára testhezálló feladatokat, a hangsúly azonban természetesen az önszervező feladat-szétosztáson van.

Fontos látni, hogy a munkát végző csapattag soha nem kap direkt utasításokat egy-egy feladat elvégzésére, nincs kiszolgáltatva a megrendelő kénye-kedvének, a követelmények csak ritkán változhatnak az ellenkezőjükre, mint az egyébkénz gyakori az ügyfélt nem íly módon kezelő módszertanok használatakor.

Mint minden munkadarab között, a *Sprint backlog item*-ek között is akárhány N:M kapcsolat hozható létre, így a tesztelési fázis során a rendszerbe felvitt hibák mindig összekapcsolhatók közvetve magával a sprinttel-, és akár a hiba felfedezése nyomán indukálódott újabb taszkokkal is.

[31] task
[32] bug
[33] impediment

A Sprint backlog item-ekhez a következő kiegészítő információkat érdemes tárolnunk:

Backlog item type – Munkadarab típusa: a három lehetséges értéke a *Bug*, a *Task*, és az *Impediment*.

Severity – Súly: ez az érték amivel jellemezhető például egy hiba komolysága, egy hátráltató tényező fontossága, avagy egy taszk prioritása.

Assigned to – Tulajdonos: a fejlesztő, aki a munkadarabért a megnyitást követően felel.

Estimated effort – Becsült költség: miután egy feladatot valaki elvállalt, az előzőleg becsült költséget a jó Scrum master felülvizsgálja a tulajdonos korábbi becsléseinek tükrében, s így előáll egy megfelelően pontos munkaóra-keret, ami a taszk elvégzéséhez biztosan elegendő.

Work remaining – Fennmaradó órák száma: értéke a sprint kezdetekor megegyezik a becsült költséggel, a fejlesztő napi rendszerességgel „farag" belőle, amikor értéke eléri a nullát, a munkadarabot lezárt állapotba kell helyezni.

Description – Leírás: a munkadarab részletes ismertetése

File attachments – Csatolmányok: hasznos lehet bug-okhoz csatolni például egy a hiba előfordulásakor a képernyőről készített pillanatfelvételt, vagy egy protokoll-implementációs taszk elvégzéséhez feltétlen szükséges lehet egy specifikációs dokumentum az adott RFC-ről.

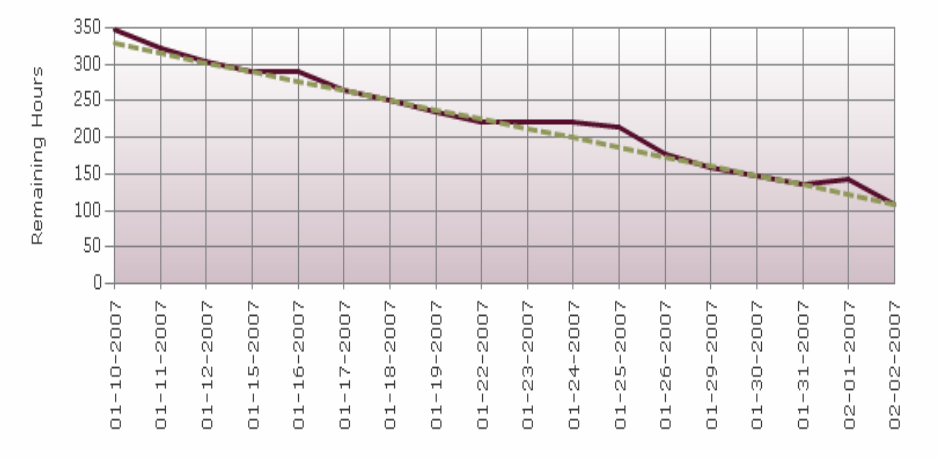
Bug details – Hiba részletei: külön fülre kerültek azok a mezők, melyeket csak a hibák leírásakor használhat a teszteléssel foglalkozó része a csapatnak. Itt adhatók meg az előfordulással kapcsolatos információk, mint például *platform*, *build száma*, *terhelési faktor*, stb. Amennyiben a nem kritikus hiba megkerülhető, a folyamat részletes leírását is elvárhatja a fejlesztő a *Workaround* nevű mezőben.

4.3. Jelentések

A Team Foundation Server két alapvető infrastruktúrális komponense az SQL Server 2005-ben található *Warehouse*-ra épülő Integration Services és az Analyzing Services. Az Integration Services valósítja meg az adatbányászatot a különböző kiegészítő tárházakból, és biztosítja a rendszeres időközönkénti mintavételt az adatbázisból. Az ilyen módon karbantartott *TfsWarehouse* nevű adatforrásra épít az Analyzing Services OLAP kockát egy sor *mérőcsoport*[34] definiálásával. Ezek között találhatók a munkadarabok módosításait, a buildek futását, vagy akár a kód változását vizsgáló-, elemző értékcsoportok. A dimenziók sokasága, és részletességük teszi lehetővé a Reporting Services számára a meglehetősen hatékony jelentések populálását.

A jelentés-generátor nem más, mint egy webes alkalmazás, amely előre definiált paraméterek alapján az RDL formátumú jelentéseket lefuttatja a TFS adatforrásán.

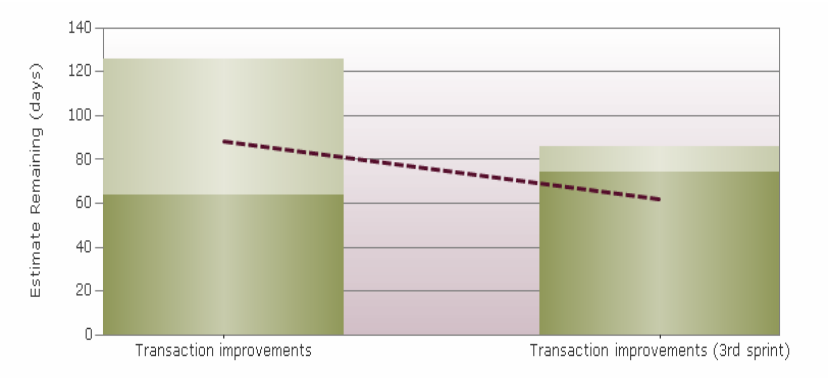
A Scrum masterek számára – és a kliens számára is – az egyik legfontosabb információ, amit mindig szem előtt kell tartaniuk; az a sprint aktuális helyzetére vonatkozó, úgynevezett *Sprint Burndown Chart*. Ez egy olyan diagram, amely a sprintbe tervezett, még hátralévő munkaórák számát ábrázolja az eltelt idő függvényében.



[34] Measure group

Mint az az ábrából is látszik, a sprint végére még maradt több mint 100 órányi feladat – tehát ez a sprint sikertelen volt. A zöld szaggatott vonal a líneáris regressziót követő állapot-előrejelzést mutatja. Amennyiben a sprint közepén ez az egyenes még mindig nem metszi a vízszintes tengelyt a sprint vége dátum előtt, a csapatnak valamilyen problémája lehet, valószínűleg külső beavatkozást igényel. Azt a sprintet is sikertelenként szokták elkönyvelni – persze csak menedzsment oldalon – , amelyiknél a taszkok hamarabb fogynak el, mint hogy az iteráció véget érne.
Az előbbiekben leírt problémák kezelésére szolgál a *Sprint Evaluation Meeting*. Ezt az elemző értekezletet rendszerint a sprint háromnegyed-részénél tartják, és az addig elért eredmények értékelésére nyújt alkalmat, esetenként válaszlépésekre adhat lehetőséget.

Az egyes sprintek lezárása után, egy termék fejlesztési menetének vizsgálatakor fontos lehet összehasonlítani az egymást követő sprintek sikerességét, valamint azt, hogy a teljes tervezett feladat-mennyiségből milyen arányban osztoztak a munka folyamán. Ehhez nyújt segítséget a *Product Burndown Chart* melyen az egyes hordók a sprinteket jelképezik, a színezésük pedig azt mutatja, mekkora részük lett befejezve.

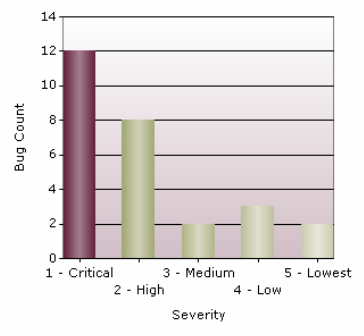


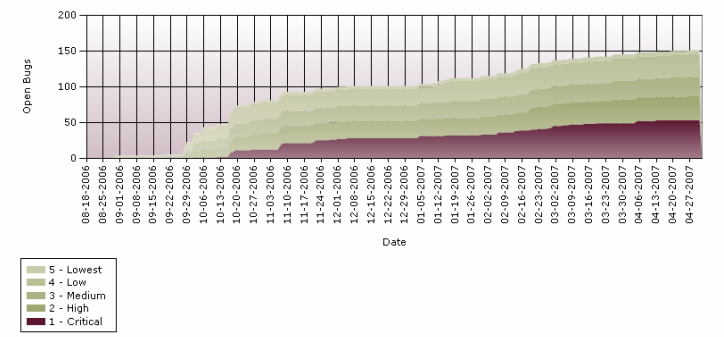
A baloldali sprint egy sikertelen sprint, a jobb oldali viszont még csak épp most kezdődött, viszont az előző kimenetele alapján szükséges lehet a team fokozott figyelemmel követésére.

Az ehhez hasonló jelentések alapján gyakran kiderülhet az, hogy nem a csapattal magával van baj, hanem az üzleti oldalal; illetve a *Product owner* folyton túl sok igénnyel él a backlogban. Az is gyakori probléma, hogy a követelmények között minden egyes prioritású. Ha képtelenek

vagyunk ezt kezelni, az a gyakorlatban úgy csapódik le, hogy a csapat a sprintbe valószínűleg nem azokat a megoldásokat sűríti be, amit valójában szeretett volna a kliens, hanem amiket a legegyszerűbben implementálhatónak ítél a sok 1-es priorítású közül. Az eredmény elégedetlenség, mindkét fél részéről.

Fontos információt szolgáltatnak a hibák eloszlásáról szóló jelentések is. A *Bug Priority Chart* és a *Bug History Chart* egyaránt színkódokat rendel a különböző komolyságú hibákhoz.





A baloldali ábra alapján látszik, hogy a csapat viszonylag ritkán vét hibát, de akkor nagyot – a kritikus hibák száma igen nagy, bár ez adódhat a tesztelést végzők szigorúságából is.
A jobboldali ábra a teljes fejlesztési intervallumot lefedi, és a rendszerben adott napon található lezáratlan hibák számát mutatja szintén súlyosság szerint.

A TFS összes projektjének adatait átfogó jelentések egyike a *Team build lista*, aminek a 3.2 fejezetben leírt kommunikációt teszi lehetővé.

Team Project Name	Build	Build Quality	Build Start time
ADE	Daily release_20070320.1	Ready for Release	3/20/2007 8:23
ADE	Daily release_20060809.1	In Production	8/9/2006 16:04
ADE	Daily release_20060804.2	Released	8/4/2006 16:03
Directory	Daily release_20060804.1	Ready for Initial Test	8/4/2006 14:51
Operations Workbench	Daily release - VWB 1.1_20070427.1	Ready for Initial Test	4/27/2007 15:12
Operations Workbench	Daily release - VWB 1.0_20070306.1	In Production	3/6/2007 12:19
Operations Workbench	Daily release - VWB 1.0_20070303.2	Released	3/3/2007 16:52
Operations Workbench	Daily release_20070216.2	Ready for Initial Test	2/16/2007 16:28
Operations Workbench	Daily release - DirectoryService_20070118.1	Ready for Initial Test	1/18/2007 9:49
Supervisor Workbench	Daily release_20061110.1	Ready for Initial Test	11/10/2006 11:05
Supervisor Workbench	Daily release_20060828.1	Released	8/28/2006 14:03
Transaction	Daily release - Client_20070402.3	Ready for Initial Test	4/2/2007 15:09
Transaction	Daily release - Server_20061218.4	Ready for Initial Test	12/18/2006 14:27
Transaction	Daily release - ClientMonitor_20061214.1	Ready for Initial Test	12/14/2006 16:05
TripTracker	Daily release_20060705.3	Released	7/5/2006 11:28
Vchat	Daily release_20061016.3	Released	10/16/2006 15:10
Vchat	Daily release_20061016.2	Ready for Initial Test	10/16/2006 13:28

Fontos szerepet töltenek be a fejlesztők munkájában a Team Explorer segítségével futtatható lekérdezések is. A sémában előre definiálható lekérdezéseket akármelyik fejlesztő saját részre lementheti, átírhatja, testreszabhatja. Mivel ugyanezek biztosítják a rendszerben található alapvető munkadarabok listázását, a Scrum masternek lehetősége van egy-egy lekérdezést a napi munkához történő optimalizálás után az egész csapata számára elmenteni.

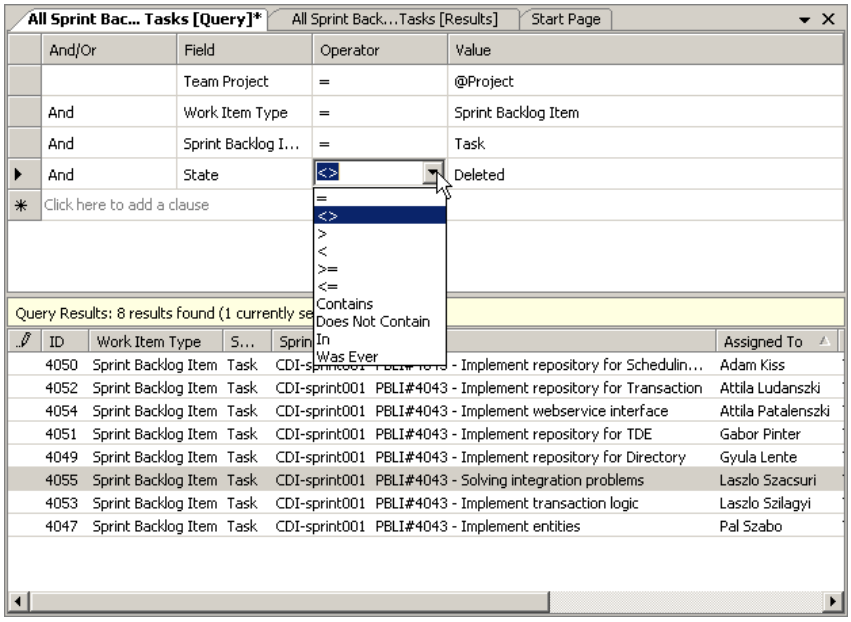
A folyamatleíró-sémánkba ágyazását ezen lekérdezéseknek a saját definiáló nyelvükkel tehetjük meg. Álljon itt egy példa a WIQL-ben rejlő lehetőségek bemutatására:

```
<?xml version="1.0" encoding="utf-8"?>
<WorkItemQuery Version="1">
      <TeamFoundationServer>http://debdev02:8080/</TeamFoundationServer>
      <WIQL>
            SELECT
                  [System.Id],
                  [System.WorkItemType],
                  [VSTS.Scrum.SprintBacklogItemType],
```

```
            [VSTS.Scrum.SprintID],
            [System.Title],
            [System.Description],
            [VSTS.Scrum.EstimatedEffort],
            [VSTS.Scrum.WorkRemaining],
            [System.AssignedTo],
            [System.State],
            [System.Reason]
      FROM
            WorkItems
      WHERE
            [System.TeamProject] = @project
      AND
            [System.WorkItemType] = 'Sprint Backlog Item'
      AND
            [VSTS.Scrum.SprintBacklogItemType] = 'Task'
      AND
            [System.State] &lt;&gt; 'Deleted'
      ORDER BY
            [VSTS.Scrum.SprintID], [System.AreaPath]
  </WIQL>
</WorkItemQuery>
```

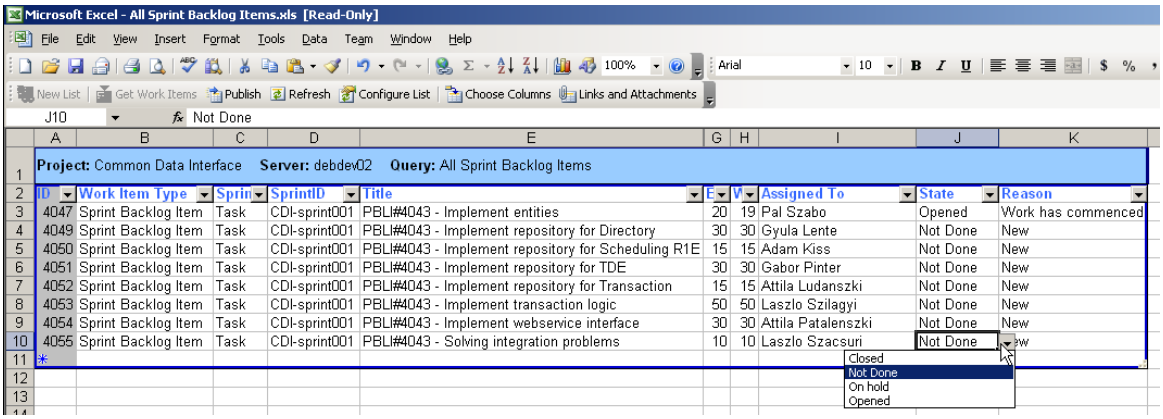
Természetesen a WIQL tagok között megadott lekérdezés egy lebutított Transact/SQL szintaktikát követ, ennek megfelelően tökéletesen illeszkedik az MS SQL Server-re épülő infrastruktúrális szolgáltatásokhoz.

A Team Explorerben található egyszerű vizuális szerkesztő segítségével ezek a lekérdezések igen könnyedén testreszabhatóak. Ez a lekérdezés például azokat a taszkokat listázza az aktuális projektünkben, amelyek nem törölt álllapotban vannak.



Egy további igazán elegáns és kényelmes szolgáltatása a Team Suite-nak az az MS Office termékcsomagba ágyazódó plug-in, melynek segítségével Excel- illetve Project dokumentumokban alkalmazhatóak ugyanazok a WIQL lekérdezések, melyekkel Visual Studioban a fejlesztők dolgoznak.

Ezt egy Product owner, aki a klienshez kiszállva offline kényszerül rögzíteni a követelményeket rendkívül jól tudja hasznosítani. Mivel az Excel tábla csak egy keret-rögzítő sémaként szolgál – viszont tartalmazza a folyamatleíró-sémánkban definiált megszorításokat is – a TFS-hez való direkt kapcsolódás nélkül vehetjük fel az elemeket. Amikor pedig végre létrehozható a kapcsolat a szerverrel, egy végellenőrzés lefuttatása után publikálhatóak a munkadarabok. Itt is működik a konkurrencia-kezelő, tehát nem fordulhat elő, hogy ha egy adott munkadarabot többen szerkesztettek offline, bármelyikük módosításai elvesszenek.



5. Összefoglalás

Ezt a dolgozatot – kevéssé titkoltan – általános útmutatónak is szántam azon fejlesztői csapatok számára, akik épp szervezetlenségből fakadó problémákra szeretnének egy jó módszertan és valamilyen eszközrendszer segítségével megoldást találni.
A tapasztalat azonban azt mutatja, hogy egy módszer alkalmazása-, és az egyes lépések betűről-betűre való betartása sem nyújthat maradéktalanul segítséget egy olyan csapatban, ahol a „játékosok" nem ugyanafelé a cél felé tartanak.

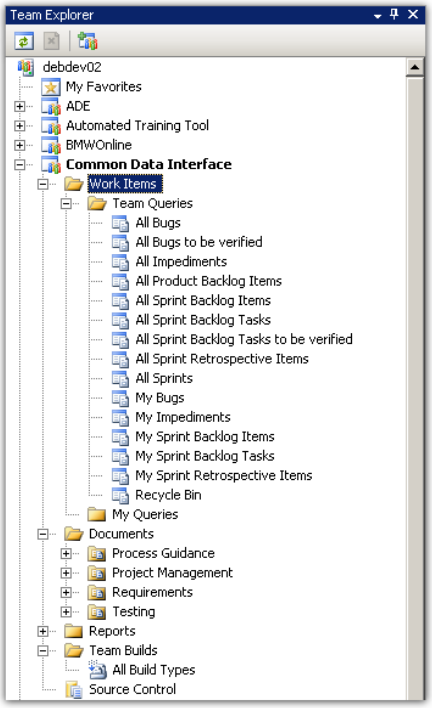
A gyakran bonyolult processzek megtanulása helyett érdemesebb mindig egy adott gondolkodási módot elsajátítanunk, és felismerni helyünket a gépezetben. Amennyiben ez a gép hangos és zakatol, az egyes részek nem feltétlenül vannak a kijelölt helyükön. Ha viszont csendes, és minden gördülékenyen megy, egyik csavar felületén sem érezhető nagyobb feszültség.
Talán a legnagyobb ismérve az olyan *agile* methodológiáknak, mint például a *Scrum*, hogy egyszerű filozófiát vallva, letisztult gondolat mentén felépített keretrendszert nyújtanak a fejlesztőknek, és szinte láthatatlanul biztosítják a legnagyobb hatékonyságot.
A szoftverfejlesztő, mint munkaerő mindig is a legnehezebben kezelhető rétegek között lesz a piacon. Kreativitása, alkotási vágya, és állandó igénye az új utak felkutatására szinte alkalmatlanná teszi arra, hogy bármilyen szabályrendszert huzamosabb ideig elviseljen.
A programcsomagot, amit igyekeztem bemutatni, számomra éppen az teszi igazán nagyszerűvé, hogy minél tovább használja az ember – a programozó – , annál tovább finomíthatja saját igényeinek megfelelően. Minél többen, minél több projektet indítanak a rendszerben, annál több hibára deríthet fényt a kollektíva, annál jobban adaptálható a rendszer a csapat szokásaihoz, módszereihez.

Márpedig ha egy jól szervezett-, szakmailag képzett csoport, melynek kellemes légkörében minden egyén jól érzi magát; jól definiált feladatokat kap, és egy belátható célt tűz ki maga elé, egyszerűen nem állíthat elő mást, mint kimagaslóan jó terméket...

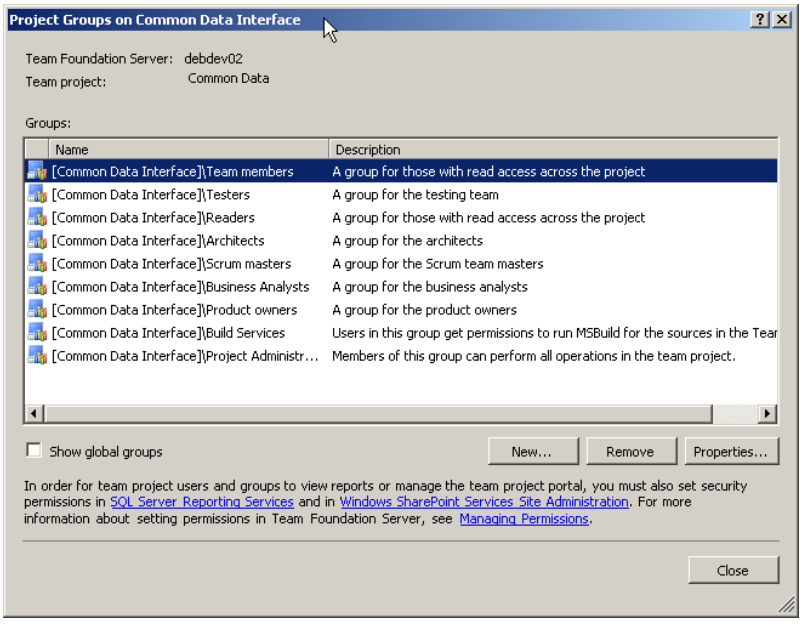
Itt szeretném megköszönni Dr. Juhász István Tanár Úrnak a dolgozat összeállítása során nyújtott segítségét, valamint a ROC Development Hungary Kft-nek, hogy engedélyezték az általam a cég fejlesztői számára kifejlesztett Scrum folyamatleíró-séma egyes részeinek publikálását.

6. Függelék

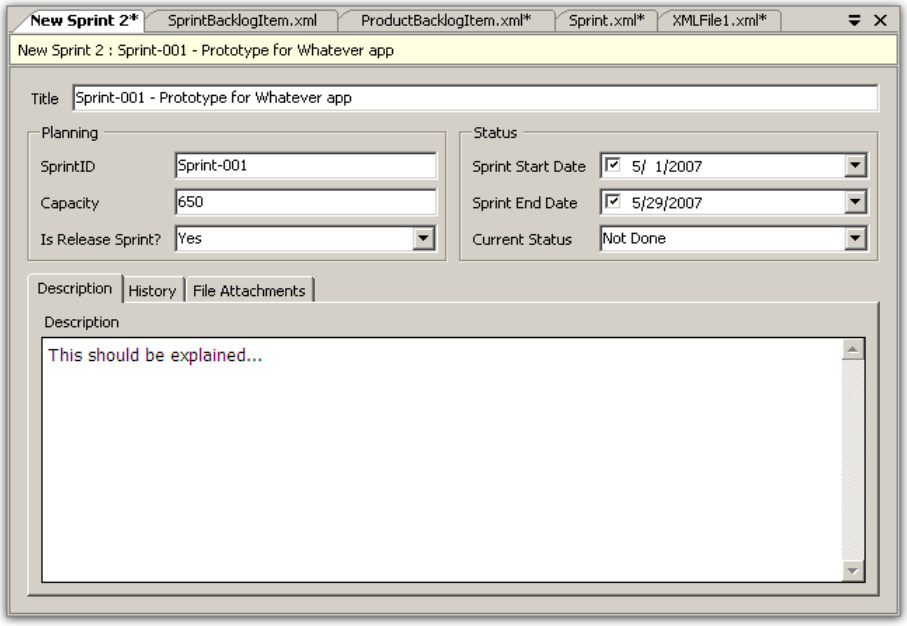
1. A Visual Studio Team Suite Team Explorer nevű pluginjának fa nézete



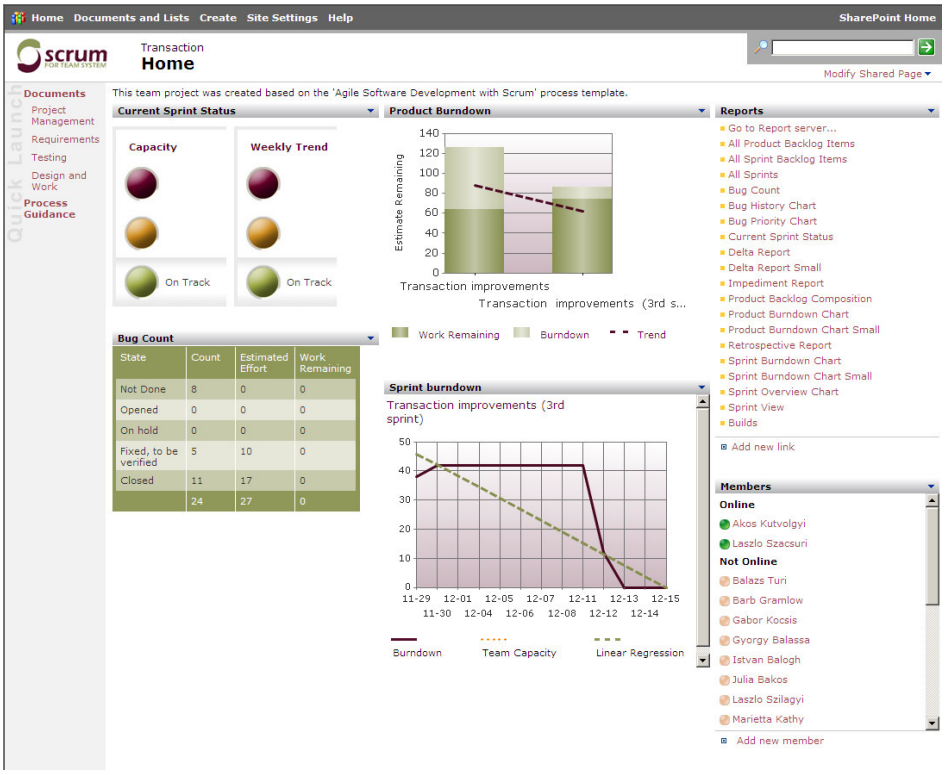
2. A TFS jogosultságkezelő felülete



3. Egy új sprint felvitele a Workitem Managerben



4. Egy projekthez tartozó sub-site a Team Portalon



7. Irodalomjegyzék

[I] Corrado Böhm, Giuseppe Jacopini.: *Flow diagrams, Turing machines and languages with only two formation rules*, CACM 9(5), 1966

[II] Edsger Dijkstra, *Notes on Structured Programming*, pg. 6, 1970

[III] Niklaus Wirth: *Program Development by Stepwise Refinement* - Communications of the ACM, Vol. 14, No. 4, 1971

[IV] Michael A. Jackson: *Principles of Program Design*, 1975

[V] W. Stevens, G. Myers, L. Constantine: *Structured Design*, IBM Systems Journal, 13 (2), 115-139, 1974

[VI] James Martin, Clive Finkelstein: *Information Engineering*, Technical Report (2 volumes), Savant Institute, Carnforth, Lancs, UK., 1981

[VII] James Martin: *Rapid Application Development*, Macmillan Coll Div, ISBN 0-02-376775-8, 1991

[VIII] Linda Rising, Norman S. Janoff: *The Scrum Software Development Process for Small Teams*, 2000

[IX] Philippe Kruchten: *The Rational Unified Process: An introduction*, 1998